ChatGPT

HITRUST i1 Readiness Audit – *HealthPrepV2* Repository

Overview: This audit evaluates *HealthPrepV2*'s codebase and documentation against HITRUST Implemented 1-year (i1) security controls. Below we examine key control areas – authentication, access control, logging, encryption, password/secret management, abuse protection, PHI handling, and compliance documentation – and assess readiness for AWS cloud deployment with encryption at rest. A gap analysis with prioritized remediation steps is provided at the end.

Authentication & Session Management

- **User Authentication:** *HealthPrepV2* implements username/password login via Flask-Login with secure password hashing (Werkzeug's PBKDF2). User passwords are never stored in plaintext; the `User.set_password` method hashes passwords before saving [1], and `check_password` verifies hashes [2]. This meets HITRUST requirements for strong credential storage. Login attempts are tracked and accounts lock after 5 failures [3] [4], preventing brute-force attacks (see Rate Limiting below). Session timeout is enforced: user accounts have a 30-minute inactivity timeout by default [5], and the app uses a 1-hour absolute session lifetime with each request refreshing the timer [6].

- **Session Security:** Sessions use secure cookies with appropriate flags. In production, `SESSION_COOKIE_SECURE=True` (HTTPS-only), `SESSION_COOKIE_HTTPONLY=True` (no JavaScript access), and `SESSION_COOKIE_SAMESITE='Lax'` to mitigate CSRF [6]. The Flask `SECRET_KEY` is required (32+ chars) for session signing and is loaded from an environment variable [7] [8]. These settings align with best practices for session management and were validated in the security checklist [9] [10].

- **Multi-Factor Authentication (MFA):** Currently, the application **requires security questions for admin users** as a second verification step on first login [11]. Admins (and root admins) must set up two security Q&A, which are stored hashed in the DB [12] [13]. This provides an extra layer of identity verification (a form of 2FA) for privileged users. However, **TOTP-based 2FA is not yet implemented** – a planned enhancement [14]. The HITRUST readiness gap summary notes that adding time-based one-time password 2FA is a future feature (priority "Low") [15].

- **Epic Hyperspace MFA:** The system integrates with Epic (EHR) via OAuth but smartly restricts which users can initiate that process. Only **"provider admin"** users (a role defined in `User.admin_type`) are allowed to perform Epic OAuth logins [16]. This design ensures that **only clinical staff with Epic access (who presumably use Epic's own MFA in Hyperspace)** can connect the app to Epic data [17]. Root administrators (superusers) are explicitly *prevented* from performing Epic SSO, adding security around that integration [17]. In practice, this means any user linking to Epic must authenticate through Epic's portal (with Epic's MFA) outside the app, meeting HITRUST requirements for external system access control.

Overall, authentication controls are strong: hashed credentials, mandatory admin security Q&A, account lockout, and secure session handling are all **implemented** [18]. The main gap is the lack of an implemented one-time-passcode 2FA for all users – which we recommend addressing to further harden authentication.

Role-Based Access Control (RBAC)

- **Defined Roles:** The application defines distinct roles with increasing privileges: standard user, admin, and root_admin (superuser). In the code, each `User` has a `role` field (e.g. "nurse", "admin", "root_admin") plus boolean flags for `is_admin` and `is_root_admin` for backward compatibility [19]. Administrators also have an `admin_type` ("business_admin" vs "provider") to differentiate those who can perform clinical (Epic) actions [20]. By default, non-admin users cannot access administrative features.

- **Access Enforcement:** RBAC is **enforced throughout** the Flask app. Flask-Login's `@login_required` protects routes, and a global `before_request` hook implements role gating. For example, if a non-admin user tries to access an `/admin/...` route, they are redirected to the user dashboard [21]. Similarly, root admins are segregated to their own interface: a logged-in root admin hitting a regular admin page will be rerouted to the root dashboard [22]. This prevents privilege escalation by URL manipulation. The HITRUST readiness mapping confirms **role-based access control is implemented** with Admin/User/Root roles in code [23].

- **Least Privilege:** Regular authenticated users can only access their own organization's data. The `User.can_access_data(org_id)` method restricts data scope – only root admins bypass this, whereas normal users must match the org_id to view data [24]. Additionally, there is logic to restrict provider-specific data access to assigned providers [25]. This design supports multi-tenant isolation: each Organization's data is segregated, and queries are scoped by org (the documentation explicitly notes **"organization-scoped queries"** for multi-tenancy isolation [26]).

- **Epic Data Access:** The system uses role checks (`can_do_epic_oauth`) to ensure only authorized provider admins can initiate the EHR integration [16]. This prevents unauthorized users from accessing external PHI via the Epic APIs.

Conclusion: RBAC controls are in place and effective. They enforce least privilege and separate duties (e.g. superuser vs admin vs user) in alignment with HITRUST access control requirements. The code and mapping docs indicate RBAC is fully **implemented** [23].

Logging & Audit Trails

- **Audit Logging:** The application maintains a detailed **audit log of administrative and security events**. An `AdminLog` model records each important action with timestamp, user, org, IP, resource, and descriptive details [27]. For instance, whenever a document is processed or viewed, or a security event occurs, a corresponding entry is added via the `log_admin_event` utility [28] [29]. This ensures a tamper-evident trail of who did what and when – critical for HITRUST and HIPAA compliance. The audit log schema even captures **patient IDs and resource types** for PHI access tracking [30] and stores user agent and session info [31] for forensic context. This meets HITRUST CSF 09.x requirements around event/audit logging.

- **Document Processing Audit:** In addition to general admin logs, *HealthPrepV2* has a **DocumentAuditLogger** component that generates structured logs for all document handling operations [32] [33]. This logger records events like "document_processed", "PHI redacted", "OCR completed/failed" along with the acting user and outcome metrics [34] [35] [36]. It uses the `log_admin_event` internally to persist these events in the audit trail [37]. Each document event thus includes who triggered it, which patient/document, and what happened (including PHI findings) [38] [39]. This granular auditing aligns with HITRUST logging requirements (e.g. ensuring **all PHI access and processing is logged** [40]).

- **Security Event Monitoring:** The system actively monitors and logs **security-specific events**. There is a `SecurityAlertService` that tracks conditions like multiple failed logins or PHI filter failures. When thresholds are exceeded (e.g. >10 failed logins from an IP in 5 minutes), it triggers alerts and logs an event. For example, on account lockout, an email alert is sent to admins and an "account_lockout" event is logged with details [41] [42]. Similarly, a "brute_force_detected" alert will be emailed and logged if many accounts are targeted by failures [43] [44]. These security alerts are integrated with the audit trail via `_log_alert_sent` calls in the service (not shown above), fulfilling incident monitoring controls (HITRUST CSF 09.1). The readiness documentation confirms **security alerting is implemented** (email notifications via Resend API) [45], including alerts for lockouts, brute force, and PHI filter errors [46].

- **Log Retention:** HIPAA requires retaining access logs for at least 6 years. *HealthPrepV2* sets an audit retention policy of **7 years** (2555 days) by default [47]. While this is configurable per organization, it meets the compliance minimum. Logs can be exported or reviewed in an admin UI (`/admin/logs`). The documentation notes a 7-year retention as implemented [48]. We recommend ensuring the production log storage (database or SIEM) enforces this retention (e.g. archival or purging beyond 7 years) in practice.

- **PHI in Logs:** The developers took care to avoid sensitive data leakage in logs. The deployment report states **"PHI-safe logging with no credential exposure"** [49]. For example, encryption errors log a generic message rather than printing secrets [50]. The `log_security_event` helper explicitly strips out any fields named 'password', 'token', or 'secret' from the details before logging [51] [52]. This attention ensures that even error/debug logs won't inadvertently contain PHI or secret values, which supports HITRUST privacy controls.

Conclusion: Comprehensive audit logging is **implemented and documented** [53]. The system logs authentication events, admin actions, data accesses, and security incidents in detail. This provides a solid foundation for HITRUST i1 compliance in audit and monitoring.

Encryption & Key Management

- **Encryption of Data at Rest:** The application uses **field-level encryption for sensitive data** in the database. Specifically, any PHI or credentials stored are encrypted using AES-256 symmetric encryption (via the Fernet library). For example, the **Epic EHR OAuth tokens and client secrets are encrypted** in the database. The `EpicCredentials` model wraps `_access_token` and `_refresh_token` with properties that encrypt on set and decrypt on get [54] [55]. Similarly, an organization's `epic_client_secret` is stored encrypted [56]. The encryption utility uses a 32-byte key from the `ENCRYPTION_KEY` env var and Fernet cipher to perform AES-128/CBC with HMAC (built into Fernet) on the data [57] [58]. The **encryption key is required in production** – if not set, the app refuses to start (as enforced by `secrets_validator`) [59]. This meets HITRUST encryption requirements by protecting sensitive fields like credentials and any PHI stored outside the primary medical data. According to project docs, **AES-256 encryption is implemented for PHI fields** [60] and was verified as part of security hardening [61].

- **Encryption in Transit:** The application is intended to be deployed behind HTTPS. The Flask app, when `FLASK_ENV=production`, will **enforce HTTPS redirects** and sets **HSTS headers** for strict transport security [62] [63]. The `utils/security_headers.configure_security_middleware` likely handles redirecting any HTTP to HTTPS and adding headers (in code it's called with `force_https=True` in prod) [64] [65]. The security checklist calls out that a reverse proxy or load balancer must forward the scheme for this to work [66]. In AWS, deploying behind an ALB with TLS will satisfy this. Thus, network traffic containing PHI will be encrypted in transit via TLS, as required. We did not find any transmission of sensitive data without encryption – even integration with Epic uses HTTPS API calls.

- **Key Management & Secrets:** The project follows strict secret management practices. **No secrets or credentials are hard-coded** in the repo – everything is supplied via environment variables [67]. A `.env.example` file documents all needed secrets, and a startup validator ensures required secrets (e.g. `SECRET_KEY`, `ENCRYPTION_KEY`, `DATABASE_URL`) are present and of correct format [8] [68]. This prevents misconfiguration from causing weak encryption (for instance, it ensures an encryption key is set in prod). The team has even authored a **Key Management Policy** document detailing generation, rotation, and storage of all secrets [69]. In that policy, each secret (encryption keys, API keys, DB URL, etc.) is classified by sensitivity and given a rotation frequency (e.g. annual rotation for critical keys like `ENCRYPTION_KEY` and `SECRET_KEY`) [70]. For example, the encryption key and session keys are to be rotated at least annually or upon suspected compromise [70]. This level of documentation and procedural detail shows strong alignment with HITRUST control 10.g on cryptographic key management.

- **Secret Storage (Current & Future):** In the current Replit environment, secrets are stored in Replit's encrypted storage and injected as env vars [71]. For the AWS migration, the plan is to use AWS Secrets Manager with a structured hierarchy for dev/staging/prod secrets [72] [73]. The documentation includes a mapping of Replit secrets to AWS Secrets Manager paths [74] and even CLI scripts for rotating secrets in AWS [75] [76]. This shows preparedness to maintain strong encryption and secret management practices in the cloud.

- **Key Rotation:** The code and docs demonstrate support for **key rotation without downtime**. The `EncryptionService` class can be re-initialized with a new key, and the team has outlined a **"dual-key" rotation procedure** for the database encryption key to re-encrypt all PHI with a new key safely [77] [78] [79]. They even include pseudo-code for a migration script to re-encrypt all records under a new key in maintenance mode [80] [81]. This indicates a mature approach to cryptographic agility, which is a HITRUST best practice. Rotation of less sensitive keys (API keys, etc.) is also documented (e.g. how to update Stripe or Resend API keys and revoke old ones) [82] [83].

- **Database Encryption at Rest:** Beyond application-level encryption of fields, the deployment guide **recommends enabling storage encryption at the database level**. For AWS RDS, the Terraform snippet provided sets `storage_encrypted = true` and attaches a KMS key [84]. This ensures the entire database volume is encrypted at rest by AWS (meeting HITRUST requirements for full disk encryption of databases containing ePHI). While this is not in code, it is clearly specified as a requirement before production [85].

Conclusion: Encryption controls are comprehensive. Sensitive fields are encrypted in the app; secrets are managed via environment with no hard-coding; TLS is enforced for transport; and plans for AWS include enabling encryption at rest. The project's design and documentation meet HITRUST i1 encryption control requirements. The main consideration is to execute the documented processes (key rotations, AWS KMS setup) in the production environment.

Password Management

- **Secure Storage:** Passwords are never stored in plaintext. The User model uses Flask/Werkzeug's PBKDF2-based `generate_password_hash`, which applies a salt and multiple iterations by default [1]. This means stored `password_hash` values are resistant to offline cracking. Verification is done with constant-time hash comparisons (`check_password_hash`) [2], preventing timing attacks. This implementation aligns with HITRUST requirements for strong password hashing (e.g., not using deprecated hashes like MD5 or unsalted SHA).

- **Password Complexity & Policies:** *HealthPrepV2* enforces a strong password policy. New passwords (including resets) must be **at least 8 characters** and meet complexity rules: at least one uppercase letter, one lowercase, one digit, and one special character [86] [87]. If a password fails any criterion, the form validation rejects it with an appropriate message [88] [89]. This ensures users choose robust passwords, satisfying HITRUST password composition standards. Additionally, default user credentials used in development (if any) have been removed – the deployment notes explicitly say the old "admin/admin123" default was eliminated [8].

- **Password Change & Reset:** The application forces users to update temporary passwords. If an account has `is_temp_password=True` (e.g. first login with a preset password), the user is redirected to a password change page before accessing anything else [90]. This prevents continued use of default or temp credentials. For password resets, a secure token mechanism is in place: `User.generate_password_reset_token()` creates a cryptographically secure random token (32 bytes url-safe) and stores a hashed version along with a 1-hour expiration [91] [92]. The user must provide the correct token (sent via email link) to reset the password, and the token expires after one use or an hour. The code also provides `clear_password_reset_token` to nullify it after use [93]. We also see that after a successful reset, the app invalidates any other outstanding tokens for that user [94]. This process aligns with HITRUST by ensuring password resets are secure and temporary credentials can't be reused.

- **Account Lockout:** As mentioned, the system locks accounts after 5 failed login attempts within a short window, for 30 minutes [3] [95]. The `failed_login_attempts` count and `locked_until` timestamp are stored per user. This is an important password security control (mitigating online guessing attacks) and is implemented according to the HITRUST guidelines (which typically suggest lockout after 3-5 attempts). The lockout triggers an alert email to admins as well [41] [42], so the event is monitored.

- **Storage of Credentials & Keys:** Apart from user passwords, other sensitive credentials (like API keys, integration secrets) are not stored in plaintext. They either reside in env vars or are encrypted if stored (as discussed in Encryption & Secrets sections). There is no evidence of any hardcoded passwords or secrets in the repo scan, which is good.

Conclusion: Password management in *HealthPrepV2* is strong. It covers secure hashing, complexity requirements [87], forced changes for temp passwords, timely reset tokens, and account lockout. These measures collectively satisfy HITRUST's authentication and password policies. We recommend implementing multi-factor for privileged logins as an enhancement, but the password controls themselves are compliant.

Secrets Management & Environment Configuration

- **Environment-Only Config:** The project demonstrates a strict practice of using environment variables for all secrets and config values. The `DEPLOYMENT_READINESS.md` report proudly notes **"Zero hardcoded secrets – all configuration via environment variables"** [67]. Indeed, in code we see things like database URLs, API tokens, encryption keys, etc., are fetched from `os.environ` [7] [96] rather than literals. A `secrets_validator` runs at startup to **fail fast** if required secrets are missing or too weak (e.g., it ensures `SECRET_KEY` is provided and `ENCRYPTION_KEY` is of correct length) [8] [68]. This prevents deployment with default or empty secrets, a critical control.

- **Configuring Secrets:** The repository includes a sample `.env.example` that lists all needed variables and proper setup instructions (not quoted here for brevity). Key variables include `SECRET_KEY`, `ENCRYPTION_KEY`, `DATABASE_URL`, and third-party API keys. The **Security Checklist** in docs provides a table of required env vars and whether they're mandatory in production [97] – reinforcing what must be set for a secure deployment. HITRUST requires management of system credentials and secrets in a secure manner, and this project clearly meets that: secrets are externalized and can be stored in a vault.

- **Secret Rotation:** As mentioned, the team has documented rotation procedures for each secret [82] [98]. Standard API keys are to be rotated annually or per vendor policy, and critical keys (encryption, session keys) annually or on compromise [70]. The AWS Secrets Manager integration described will facilitate rotation (Secrets Manager supports automated rotation for certain secrets, and manual steps are given for others) [99] [100]. The key management policy even covers a **dual-key rotation** for the encryption key, which is typically a challenging process, showing foresight in maintaining confidentiality long-term [77] [78].

- **Use of Secret Manager (AWS):** In preparation for cloud, secrets will reside in AWS Secrets Manager with distinct namespacing for environments (dev/staging/prod) [73] [74]. The app will retrieve them via the environment (e.g. by injecting through ECS task definitions as shown [101] [102]). This design keeps secrets out of code and out of unencrypted config files, aligning with HITRUST's key protection requirements.

- **No Exposure in Repositories:** We did not find any API keys, passwords, or certificates checked into the repository history. The `.gitignore` likely covers any local `.env` files. This reduces risk of secret leakage and is a HITRUST-friendly practice.

Conclusion: Secret management is handled in a robust, compliant way. The project has policies and code enforcement to ensure secrets are properly generated, stored, and rotated. The move to AWS with Secrets Manager will further enhance this by removing reliance on Replit's store. These practices align well with HITRUST i1 control requirements for secure configuration management and key protection.

Rate Limiting & Abuse Protections

- **Rate Limiting:** *HealthPrepV2* implements **IP-based rate limiting** on sensitive endpoints to thwart abuse (e.g. brute force attacks, denial of service). The `utils/security.py` module defines a `RateLimiter` class with configurable thresholds for different endpoints [103]. For instance, by default: login attempts are limited to 5 per 5 minutes with a 30-minute lockout [103], password reset requests 5 per 5 minutes, security question answers 3 per 5 min, etc. These limits are in line with OWASP guidelines and reduce the risk of automated attack or spam. The readiness checklist confirms **IP-based rate limiting is implemented** with a 5-attempt lockout on login [104].

- **Distributed Environment Consideration:** The rate limiter is designed to support multi-instance deployment. It will use Redis if available for a shared rate limit store, otherwise it falls back to an in-memory dict for single-instance scenarios [105] [106]. Warnings are logged if Redis is not configured in production, since without it an attacker could bypass limits by switching instances [107]. This shows the developers have considered the production cloud environment and the need for a central throttling mechanism – a positive sign for cloud readiness. (We recommend enabling Redis or another centralized cache in the AWS deployment to utilize this.)

- **Implementation in Routes:** The `rate_limit` decorator is applied to routes like login and password reset flows. In the code, before processing a request, it checks `RateLimiter.check_rate_limit(...)` and if not allowed, it flashes a "Too many attempts, wait X minutes" message and blocks the action [108]. This user feedback and throttling mechanism is user-friendly while effective. The Security Checklist highlights that rate limiting covers login, password reset, and security question verification attempts [109] and supports Redis for distribution [110].

- **Brute Force Detection:** In addition to simple rate limiting, the SecurityAlertService monitors for broader brute force patterns. If more than 10 failed logins from the same IP occur within 5 minutes (configurable threshold) [43], it triggers a **"brute_force_detected"** alert email to the security contacts [44] [111]. This alert includes the IP, number of attempts, and usernames targeted [112] [113], enabling administrators to take action (like IP blocking or notifying users) [114]. Logging of this event also occurs, as noted earlier. This aligns with HITRUST incident detection requirements – not just preventing brute force via lockout, but actively detecting and responding to it.

- **Denial of Service (DoS):** While not explicitly mentioned, the presence of rate limiting inherently provides some DoS mitigation for heavy endpoints. The app's design (Flask + SQL) would rely on external infrastructure (like an AWS WAF or load balancer) for volumetric DDoS protection. The deployment guide indeed recommends using an AWS WAF on the ALB [85]. Application-level, there are no obvious expensive operations exposed without authentication – most endpoints are internal or require login. We did see asynchronous tasks for document processing, but those are triggered by user actions (which are limited by login).

- **Other Abuse Protections:** The content security policy (CSP) headers and CSRF protection are enabled to guard against XSS/CSRF abuses (see PHI Handling section for CSP). Flask-WTF CSRFProtect is initialized in the app [115] [116] and likely applied globally (we saw `csrf.init_app(app)` in create_app [117]). This prevents unauthorized cross-site POSTs, which is part of abuse prevention in a broader sense.

Conclusion: The project has effective throttling and monitoring to prevent and detect abuse. Rate limiting is in place for authentication flows [118], and security alerts ensure administrators are aware of suspicious activities. For HITRUST i1, these controls demonstrate due diligence against brute force and abuse scenarios. We recommend, in production, leveraging the existing support for Redis or using AWS's API Gateway/Load Balancer features to further enforce these limits globally.

PHI Handling & Data Protection

- **PHI Minimization:** The system is built to handle Protected Health Information (PHI) (patients, medical documents, etc.), and it incorporates **data minimization and tagging to reduce PHI exposure**. For instance, document types are identified by LOINC codes (standard medical codes) rather than storing verbose PHI-laden descriptions [119]. This suggests that wherever possible, the app uses coded values instead of full PHI text. The code also has a setting `phi_logging_level` per organization (default "minimal") [47] which likely controls how much PHI (if any) is included in logs – by default minimal means it will avoid logging full patient data.

- **PHI Redaction:** A robust **regex-based PHI filter** is implemented to scrub sensitive identifiers from text. The `ocr/phi_filter.py` module contains extensive regex patterns for SSNs, phone numbers, emails, MRNs, insurance IDs, financial info (credit cards, account #s), government IDs, provider IDs (NPI, DEA), addresses, etc. [120] [121] [122]. When document text is processed (likely after OCR), the filter replaces detected PHI with placeholder tags like "[SSN REDACTED]", "[PHONE REDACTED]", etc. [123] [124] [125]. It carefully avoids double-redacting already redacted text by recognizing "[... REDACTED]" tokens [126] and not altering them. It also tries to **preserve medical terms and necessary context** (to not over-redact important health information) by protecting certain spans (like "blood pressure 120/80" should not redact "120/80" as it's not an ID) [127] [128]. This PHI filter is applied consistently to any text output or stored from documents. The presence of a PHI filtering mechanism is explicitly noted as implemented in the HITRUST readiness doc [119] and the security checklist lists all categories of PHI it covers [129]. This control is crucial for HITRUST – it ensures that if documents are stored or shown, direct identifiers can be removed or masked, thereby limiting exposure.

- **Secure Data Deletion:** The application accounts for secure deletion of files containing PHI. There is a `utils/secure_delete.py` (mentioned in docs) which likely overwrites and deletes temporary files used in processing. Indeed, on startup the app calls `cleanup_registered_temp_files()` to remove any leftover temp files from crashes [130], logging the cleanup of "orphaned temp files" as a **HIPAA safeguard** [131]. This is to prevent PHI from lingering in the file system. The readiness checklist notes *"Secure file deletion – 3-pass overwrite – Implemented"* [132], indicating that whenever PHI files are deleted, they are overwritten multiple times (three passes) to mitigate forensic recovery. This is an advanced control showing a high level of PHI care (and aligns with HITRUST/HIPAA requirements for proper disposal of ePHI).

- **No PHI in Logs or URLs:** The code and config aim to ensure PHI does not accidentally leak via logs or referrer URLs. For example, any PHI that might be part of an URL query param is covered by the PHI filter (`phi_urls` patterns) [122] – so if a link with patient info query string were present, it would be redacted. Logging, as mentioned, is PHI-safe. Also, by using numeric IDs and codes, URLs likely reference internal IDs rather than patient names, etc. The CSP and Referrer-Policy header (`strict-origin-when-cross-origin` is set [133]) further ensure that if any page with PHI links out, the full URL (which might contain an ID) isn't shared externally.

- **Data Integrity & Quality:** While not explicitly asked, it's worth noting that all input forms use server-side validation (Flask-WTF) including data type checks and CSRF tokens. This prevents malformed or malicious data from contaminating the system. There's also mention of "Input validation, XSS prevention, SQL injection prevention" all being implemented (using WTForms, auto-escaping templates, and SQLAlchemy ORM respectively) [134]. This indirectly protects PHI by ensuring that the system's data isn't corrupted or exposed via injection attacks.

- **Audit of PHI Access:** Every access to PHI (like viewing a patient or document) is logged via the AdminLog as described, including which patient record was accessed [30]. This provides an audit trail to detect any unauthorized snooping on patient data, a HITRUST requirement.

Conclusion: PHI handling is a standout strong area for this project. The combination of **PHI redaction, minimal necessary data storage, secure deletion, and strict logging** demonstrates a design built around HIPAA compliance. These controls map directly to HITRUST CSF privacy controls and appear **fully implemented** [135]. The team should continue to monitor and tweak the regex patterns to avoid false negatives/positives, but overall PHI is well protected in this system.

Compliance Documentation & Policy Evidence

- **Policies and Procedures:** The repository isn't just code – it includes extensive documentation of security policies and compliance evidence. Notably, there is a **HITRUST CSF Readiness Checklist** document that maps the application's controls to HITRUST domains [136]. This checklist shows each requirement, how HealthPrep addresses it, where evidence is located (code references), and status [137] [53]. Such a mapping will be extremely valuable in a HITRUST i1 audit. For example, it cites the **Information Security Policy** (to be in a SECURITY_WHITEPAPER.md), Data classification (PHI classification in code via phi_filter.py), Encryption requirements (AES-256 in models), etc., marking them as *Implemented* [137] [60].

- **Formal Security Documents:** The repo contains specific policy docs, including:

- **Incident Response Plan:** A documented IR plan is present (`docs/INCIDENT_RESPONSE_PLAN.md`) and marked as implemented [138]. This would outline steps to handle security incidents, breach notification (the code even tracks the 60-day breach notification window as an event [139]), etc. Having this is crucial for HITRUST i1, which requires incident management procedures.
- **Risk Management:** There is a risk assessment register following NIST 800-30 (`NIST_800_30_RISK_REGISTER.md`), listed as implemented [140]. This indicates the team has identified threats and mitigations, aligning with HITRUST requirement to perform periodic risk assessments.

- **Key Management Policy:** As we saw, a comprehensive Key Management Policy exists [69], aligning with HITRUST and HIPAA requirements for handling encryption keys and other secrets.
- **Security Checklist / Hardening Guide:** The `SECURITY_CHECKLIST.md` [141] and `DEPLOYMENT_READINESS.md` [142] serve as evidence of security hardening tasks completed and those pending. These cover many controls (CORS config, use of TLS, WAF, backups, etc.) and show management of those items.

- **Business Continuity/Backup:** There are notes in docs about backup strategy (e.g., AWS RDS Point-In-Time-Recovery and data retention configurable per org) [143], though a full Business Continuity/ Disaster Recovery Plan document might still be needed (it's listed as not yet provided in the checklist [144]).

- **AWS Migration Planning:** The documentation evidences thorough planning for the migration to AWS, which itself is part of compliance (ensuring the new environment meets security reqs). The AWS Migration Guide in `DEPLOYMENT_READINESS.md` covers infrastructure as code, network security, CI/CD, testing, and cost estimates [145] [146] [147]. It explicitly recommends components like encrypted RDS, private subnets, WAF, CloudTrail logging, signing a Business Associate Agreement (BAA) with AWS [85] [148], etc. This level of detail will serve as evidence that the team considered and planned controls in the cloud environment as well.

- **Vendor Management (Epic & AWS):** The risk register or policy likely touches on vendor risk (Epic's APIs and AWS as a hosting provider). The checklist mentions needing a **BAA with AWS** [148] and likely tracking Epic's compliance since patient data flows through Epic's sandbox/production API. Ensuring those agreements are in place is required for HITRUST; it appears the team is aware (since it's listed as a to-do).

- **Development Practices:** The repository's organization (with automated security audit scripts [149], CI integration suggestions [150], etc.) shows a culture of security in the SDLC. For instance, they include a Python `security_audit.py` script that can be run to check all these controls, and even suggest running it in CI to block deployments if issues are found [151]. This is an excellent practice demonstrating continuous compliance verification.

Conclusion: The presence of these documents and checklists demonstrates **strong compliance maturity**. HITRUST certification requires not just technical controls but also policy and process – the project has already drafted or implemented many of these (IR plan, key policy, risk assessment, etc.). A few administrative items (e.g., formal BCP, training records) remain to be completed [144], but overall the documentation evidence is well-prepared for an i1 assessment.

AWS Cloud Migration Readiness (Encryption at Rest & Cloud Controls)

The project is currently running in a Replit environment but is **largely cloud-agnostic and ready for AWS** deployment. In fact, the documentation explicitly states the design is cloud-agnostic and prepared for migration [67] [152]. Here's how it fares regarding AWS and encryption at rest:

- **Application Architecture:** The Flask application and PostgreSQL database are configured via environment variables, so moving to AWS (e.g., AWS RDS for Postgres, ECS or EC2 for Flask) requires minimal code changes. The code already handles differences in dev vs prod (e.g., forbidding SQLite in production) [96] [153]. It also logs connection info and would warn if the DB host couldn't resolve – showing it's ready for a cloud DB endpoint [154]. Static files and user-uploaded documents would need a storage solution like S3; the team has not hardcoded any filesystem paths that would impede using S3 (though we'd check that documents are either in DB or ephemeral storage). The guide does mention using S3 for document storage with encryption enabled [85].

- **AWS Security Best Practices:** The deployment guide enumerates key AWS security measures:

- Use of **private VPC subnets** for app and DB servers, and only ALB in public subnet [146] [147] (principle of least exposure).
- **Encryption at rest** for RDS (as noted, `storage_encrypted=True` and a KMS key) [84], and for S3 (default bucket encryption) [85].
- **Web Application Firewall (WAF):** recommeded on the ALB to filter malicious traffic [85].
- **CloudTrail & CloudWatch:** enabling CloudTrail for auditing AWS API activity and CloudWatch for logs/metrics is mentioned as part of HIPAA compliance [155].
- **BAA with AWS:** as noted, they know to sign the Business Associate Agreement, which is required before hosting PHI on AWS [148].

- **Backups and DR:** suggestions for automated snapshots and cross-region backups are included [156].

- **AWS Services Integration:** The documentation provides example configurations, e.g., sample AWS CLI commands to store secrets [157] [158] and a snippet of a GitHub Actions CI pipeline to build/deploy to ECS [159] [160]. This indicates the project maintainers have a concrete plan for cloud deployment and continuous delivery, increasing confidence that the transition will be smooth and secure.

- **Encryption at Rest Readiness:** Aside from application-level encryption (already in place), the app will rely on AWS-managed encryption for full volumes and S3. This is straightforward to enable (just configuration, no code changes). Given the app's heavy use of env vars and Secrets Manager integration, keys for encryption are already externally managed (so we can easily use KMS to manage those secrets). The biggest task will be to ensure that any data stored in AWS (DB, S3, EBS volumes if any) has encryption toggled on. The Terraform example ensures RDS uses a KMS key [84]. We also recommend generating a specific KMS CMK for HealthPrep's use (for audit control over key usage), and the key management policy would need to cover managing that AWS KMS key (rotation, etc., though AWS can auto-rotate CMKs yearly).

- **Identity and Access Management:** Not explicitly covered in code, but for AWS one should enforce least privilege IAM roles for the application (e.g., ECS task role only allowed to read needed secrets from Secrets Manager, not list all secrets). The Secrets Manager policy snippet provided shows a tight scope (only resources under `/healthprep/*`) [161] [162]. This is a good sign of applying least privilege in the cloud context.

- **Monitoring and Response in AWS:** The plan to use CloudWatch and alerts for things like failed login spikes [163] will complement the application's own alert emails. Combining CloudTrail log monitoring (for unusual AWS console or API activity) with the app's audit logs will give a full picture. All these are mentioned as to-dos, which is appropriate at this stage.

Conclusion: The project is well-prepared for AWS migration. The team has identified the necessary AWS controls (encryption at rest, network isolation, secret vault, monitoring) and baked those into their readiness plans. To achieve HITRUST i1 alignment in the cloud, the main steps are **executing that plan**: deploy to AWS with the recommended architecture, enable encryption on all storage, configure AWS logging (CloudTrail), and establish the required compliance paperwork (BAA). There do not appear to be technical blockers in the code for this; it's a matter of infrastructure configuration. Once on AWS, the system will benefit from additional security layers (WAF, Security Groups, etc.) that, combined with the strong app-level security, should meet HITRUST i1 requirements for a hosted environment.

Gap Analysis and Next Steps

HealthPrepV2 exhibits a high level of security readiness, with many controls already implemented or planned. Remaining gaps are relatively minor and can be addressed to ensure full HITRUST i1 compliance. Below is a summary of gaps identified and **prioritized recommendations** to close them:

1. **Implement Time-Based 2FA for User Logins (High Priority):** The absence of TOTP or push-based two-factor authentication for standard logins is a notable gap. While admin security questions are a form of 2FA, they are weaker than one-time codes. **Next Step:** Integrate a TOTP 2FA (e.g., Google Authenticator or Twilio Verify) for all user accounts, at least as an optional setting. This was noted as a future feature [14] – accelerating it will strengthen authentication and meet HITRUST multi-factor requirements for remote access.

2. **Complete and Test the Incident Response Plan (High Priority):** The IR Plan document exists [164], but the team should conduct a **tabletop exercise** or simulation to test it (as they marked "Formal incident response testing" as needed [165]). **Next Step:** Perform an IR drill (e.g., simulate a breach of a test environment) and document lessons learned. This will satisfy HITRUST requirements for incident response *testing* and ensure the team is prepared for real incidents.

3. **Conduct a Third-Party Penetration Test (High Priority):** A penetration test is planned but not yet done [166]. This is crucial for uncovering any unseen vulnerabilities. **Next Step:** Engage a certified third-party to perform a penetration test and a vulnerability scan of the application (in its staging or AWS environment). Track and remediate any findings. This will fulfill HITRUST vulnerability management and penetration testing expectations.

4. **Harden Content Security Policy – Remove `unsafe-inline` (Medium Priority):** Currently, the CSP allows some inline scripts (for compatibility). The readiness doc lists CSP hardening as a gap (remove unsafe-inline) [167]. **Next Step:** Refactor any inline JavaScript in templates to external files and enable a stricter CSP (the app has a `CSP_STRICT_MODE` flag available [168]). This will mitigate XSS risk further and demonstrate defense-in-depth.

5. **Ensure Distributed Rate Limiting (Medium Priority):** Make sure to configure Redis (or an equivalent) in production so that rate limiting is global across all app instances [105] [106]. Without it, an attacker could target different servers to evade limits. **Next Step:** In AWS, use Amazon ElastiCache (Redis) or DynamoDB to back the rate limiter. This aligns with the documentation's recommendation [169] and will close a potential loophole in abuse prevention.

6. **Finalize Remaining Documentation (Medium Priority):** A couple of administrative documents need completion: the **Business Continuity/Disaster Recovery Plan** and **Workforce Security Training records** are listed as not done [144]. **Next Step:** Develop a simple BCP/DR plan (covering RPO/RTO, backup restoration procedures – much is already hinted in docs) and ensure all team members have completed HIPAA/HITRUST training with records. Having these will check off HITRUST i1 requirements around contingency planning and training.

7. **Sign AWS BAA and Enable CloudTrail (Medium Priority):** Before going live on AWS, execute the Business Associate Agreement with AWS (if not already) [148]. Also enable CloudTrail logging for all regions to capture AWS API activity [155]. **Next Step:** Treat the BAA signing as a gating item for production. Set up CloudTrail and configure log retention (the AWS account should have logs going to an S3 bucket with encryption). This satisfies regulatory obligations for cloud service usage.

8. **Monitor Dependency Vulnerabilities (Low Priority):** The dependency update process is "in progress" [170]. To strengthen this, consider using GitHub Dependabot or a similar tool continuously. **Next Step:** Implement automated alerts for library vulnerabilities and patch regularly. This ongoing task supports HITRUST's risk management domain.

9. **Privacy and Consent (Low Priority):** Ensure that if any patient data originates from outside sources, proper consent and data use agreements are in place. (This is more of a procedural gap – the technical controls for privacy are strong as noted.)

10. **Continuous Improvement:** After migration to AWS, perform a HITRUST self-assessment or use a readiness tool to double-check compliance against the latest HITRUST i1 requirement list. Address any minor gaps found. Keep all documentation (policies, network diagrams, data flow diagrams) up to date as the infrastructure changes.

By addressing the above items – especially the high priority ones like 2FA and testing – *HealthPrepV2* will close its remaining gaps. The **good news** is that the core security controls are already in place or well-documented. The project shows a clear commitment to security by design, and with the recommended next steps, it should be well-positioned to achieve HITRUST i1 certification.

Sources:

- HealthPrepV2 HITRUST Readiness Checklist 18 53 135 14
- HealthPrepV2 Security Checklist and Deployment Docs 171 61 49 85
- HealthPrepV2 Code (User model, Auth, Encryption, Audit, PHI Filter, etc.) 1 6 27 120

---

1 2 3 4 5 12 13 16 17 19 20 24 25 27 28 29 30 31 47 54 55 56 91 92 93 95 models.py
https://github.com/mitfusco98/HealthPrepV2/blob/4808071c7f92ea121bde76137a6e3175d2415083/models.py

6 7 11 21 22 64 65 90 96 115 116 117 130 131 153 154 app.py
https://github.com/mitfusco98/HealthPrepV2/blob/4808071c7f92ea121bde76137a6e3175d2415083/app.py

8 49 50 61 62 63 67 68 84 85 101 102 142 145 146 147 152 155 156 157 158 159 160 163 DEPLOYMENT_READINESS.md
https://github.com/mitfusco98/HealthPrepV2/blob/4808071c7f92ea121bde76137a6e3175d2415083/DEPLOYMENT_READINESS.md

9 10 40 66 97 109 110 129 141 149 150 151 168 169 171 SECURITY_CHECKLIST.md
https://github.com/mitfusco98/HealthPrepV2/blob/4808071c7f92ea121bde76137a6e3175d2415083/docs/SECURITY_CHECKLIST.md

14 15 18 23 26 45 46 48 53 60 104 118 119 132 134 135 136 137 138 139 140 143 144 148 164 165 166 167 170 HITRUST_READINESS.md
https://github.com/mitfusco98/HealthPrepV2/blob/4808071c7f92ea121bde76137a6e3175d2415083/docs/HITRUST_READINESS.md

32 33 34 35 36 37 38 39 document_audit.py
https://github.com/mitfusco98/HealthPrepV2/blob/4808071c7f92ea121bde76137a6e3175d2415083/utils/document_audit.py

41 42 43 44 111 112 113 114 security_alerts.py
https://github.com/mitfusco98/HealthPrepV2/blob/4808071c7f92ea121bde76137a6e3175d2415083/services/security_alerts.py

51 52 94 103 105 106 107 108 133 security.py
https://github.com/mitfusco98/HealthPrepV2/blob/4808071c7f92ea121bde76137a6e3175d2415083/utils/security.py

57 58 59 encryption.py
https://github.com/mitfusco98/HealthPrepV2/blob/4808071c7f92ea121bde76137a6e3175d2415083/utils/encryption.py

69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 98 99 100 161 162 key-management-policy.md
https://github.com/mitfusco98/HealthPrepV2/blob/4808071c7f92ea121bde76137a6e3175d2415083/docs/security/key-management-policy.md

86 87 88 89 password_reset_forms.py
https://github.com/mitfusco98/HealthPrepV2/blob/4808071c7f92ea121bde76137a6e3175d2415083/forms/password_reset_forms.py

120 121 122 123 124 125 126 127 128 phi_filter.py
https://github.com/mitfusco98/HealthPrepV2/blob/4808071c7f92ea121bde76137a6e3175d2415083/ocr/phi_filter.py